Non-functional properties in the model-driven development of service-oriented systems

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Non-Functional Properties in the Model-Driven Development of Service-Oriented Systems

Stephen Gilmore[1], **László Gönczy**[2], **Nora Koch**[3,4], **Philip Mayer**[3], **Mirco Tribastone**[1], **Dániel Varró**[2]

[1] University of Edinburgh, UK

[2] Budapest University of Technology and Economics, Hungary

[3] Ludwig-Maximilians-Universität München, Germany

[4] Cirquent GmbH, Germany

Received: date / Revised version: date

Abstract Systems based on the Service-Oriented Architecture (SOA) principles have become an important cornerstone of the development of enterprise-scale software applications. They are characterized by separating functions into distinct software units, called services, which can be published, requested and dynamically combined in the production of business applications. Service-oriented systems (SOSs) promise high flexibility, improved maintainability, and simple re-use of functionality.

Achieving these properties requires an understanding not only of the individual artifacts of the system but also their integration. In this context, non-functional aspects play an important role and should be analyzed and modeled as early as

possible in the development cycle. In this paper, we discuss modeling of non-functional aspects of service-oriented systems, and the use of these models for analysis and deployment.

Our contribution in this paper is threefold. First, we show how services and service compositions may be modeled in UML by using a profile for SOA (UML4SOA) and how non-functional properties of service-oriented systems can be represented using the non-functional extension of UML4SOA (UML4SOA-NFP) and the MARTE profiles. This enables modeling of performance, security and reliable messaging. Second, we discuss formal analysis of models which respect this design, in particular we consider performance estimates and reliability analysis using the stochastically-timed process algebra PEPA as the underlying analytical engine. Last but not least, our models are the source for the application of deployment mechanisms which comprise model-to-model and model-to-text transformations implemented in the framework VIATRA. All techniques presented in this work are illustrated by a running example from an eUniversity case study.

Key words Non-functional Properties, Service-Oriented Software, SOA, Modeling, Model-Driven Engineering

1 Introduction

Service-oriented computing (SOC) focuses on the development and integration of distributed, interoperable systems based on a set of autonomous, platform-independent units called services. Service-orientation aims at a loose coupling of

these services by means of orchestration of services, i.e. combining and re-using the services in the production of business applications. These characteristics have now pushed service-oriented systems towards widespread success, demonstrated by the fact that many large companies have invested a lot of effort and resources in promoting service delivery on a variety of computing platforms, mostly in the form of Web services. Very soon there will be a plethora of new services for e-government, e-business, and e-science, and other areas within the rapidly evolving Information Society, leading to a pressing demand for effective techniques and automated methods for engineering service-oriented systems.

A range of domain-specific languages and standards are already available for engineering service-oriented architectures (SOAs), such as WSDL, BPEL, WCDL, WS-Policy, and WS-Security. These deal with the various artifacts of SOA systems, such as service descriptions, orchestrations, policies, and non-functional properties at specification level. However, more systematic and model-based approaches for the development of service-oriented systems (SOSs) are still in their infancy. Most of the proposed modeling languages focus on the structural aspects of services [26, 24, 19, 6, 32]. Some represent low-level service constructs, e.g. BPEL [32] and only a few translate the models into platform-independent models (PIMs) or platform-specific models (PSMs) (e.g. [6, 32]).

Achieving the properties of service-oriented systems mentioned above requires instead (1) an understanding of the individual artifacts of the system, their specification and their integration – in other words, a complete picture of the system represented at a high level of abstraction, (2) techniques for the early estimation

and evaluation of quality of service, and (3) mechanisms for the automated generation of applications. Model-driven development (MDD) methods are the most appropriate approaches to support both specification (i.e. 1) and generation (i.e. 3) of SOA software, and to ease model-based quantitative and qualitative analysis.

In this paper, we present an MDD-based approach to modeling, analysis, and deployment of service-oriented systems, and focus on how to deal with non-functional properties such as performance, security and reliable messaging. This MDD process consists of a chain of model transformations which start with the models of the application and produce platform-independent models (so-called PIMs in the MDA terminology) and generate platform-specific models (PSMs in MDA) by PIM2PSM mappings. Some of the non-functional aspects can be directly implemented by using WS-standards (e.g., reliable messaging, security, logging, etc.) while others are effected by the underlying system architecture (e.g. performance). Therefore we target the first group by automated deployment mechanisms based on standards while the latter is the subject of quantitative analysis.

The approach is based on a profile for modeling services in UML, called UML4SOA [34,33], which we extend to include very generic non-functional specifications which are bound "per contract" to the services in the structural model (we call this extension UML4SOA-NFP). For modeling the quantitative behavior of service-oriented systems, we opted for using the specifications offered by the OMG MARTE (Modeling and Analysis of Real-time and Embedded systems) profile [43]. A UML profile is a light-weight extension of the UML frequently used to define a domain-specific modeling language, which is performed using

the extension mechanisms the UML itself offers, i.e. stereotypes, tagged values and constraints. MARTE deals with concerns of model-based analysis across the spectrum from specification to detailed design of real-time and embedded systems. MARTE facilitates the annotation of models with information required to perform specific analysis. Specifically, MARTE focuses on performance and schedulability analysis. The models built using the UML4SOA and MARTE profile are used on the one hand as source for a set of model-to-model and model-to-code transformations ending up with the deployment of the service-oriented system. On the other hand, they allow for software evaluation at design time providing e.g. a performance analysis in an early phase of the software development. Our model-driven approach and the analysis techniques are fully tool-supported.

The paper is structured as follows: In Section 2 we present the characteristics of the development process of service-oriented software; we use a scenario of a distributed eUniversity course management system to illustrate the challenges in the development of this kind of software. This scenario is also used as the running example in the remaining sections. Section 3 describes the UML4SOA modeling approach focusing on the functional aspects. Section 4 presents the extension of UML4SOA for covering non-functional aspects of service-oriented systems (UML4SOA-NFP). The extension comprises model elements for specifying non-functional properties in structural and behavioral UML diagrams. Section 5 shows how software analysis methods can evaluate the models built with the profiles presented in the previous sections to realize a performance analysis. Section 6 completes the development process presenting model-driven deployment mech-

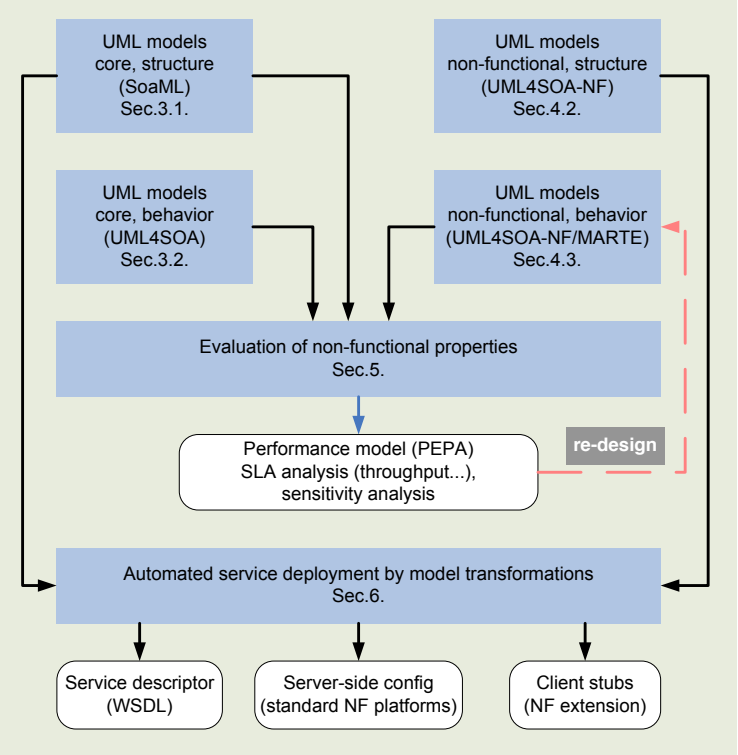

Fig. 1 Overview of the approach

anisms. We compare our approach to related work in Section 7. Some conclusions and the next steps in our research on the model-driven development approach for SOAs are presented in Section 8. Fig. 1 gives an overview of the main contributions of this paper for the model-driven development of service-oriented systems and shows the role of the non-functional aspects in our approach.

2 Challenges in the Development of Service-Oriented Systems

A Service-Oriented Architecture separates functions into distinct software units called services which users can combine and reuse in the production of business

applications. Service descriptions are published by service providers and services are invocable by a service requester according to a set of access policies. The service interface describes the set of interactions supported by a service. Service-orientation aims at a loose coupling of these services by means of the orchestration of services, i.e. the description of an executable pattern of invocations that must be followed in order to automatically coordinate, manage and arrange the set of services. An orchestration is in our approach also defined as a service.

The advantages offered by service-oriented software once in production have their costs in the development phase as complexity increases due to the additional orchestration, compensation, publishing of services, and management of service-level agreements which need to be addressed. There are primarily two important software engineering mechanisms which address the problem of increasing complexity and offer mechanisms to ease development. On the one hand, domain-specific languages, in particular domain-specific modeling languages (DSML), focus on the concepts used in a domain, which are of greatest significance for the work in a specific area. Very often a concept is a pattern-like feature which allows the users of the languages to reduce the complexity of code or models. On the other hand, automatic code generation based on models – i.e. model-driven development – eases the production and maintenance of software. Of course, appropriate tool support is required for modeling, transforming the models, and generating code.

Model-driven development makes models predominant artifacts of the development and emphasizes the automation of the engineering process. Cornerstones of the MDD approaches are modeling languages for the specification of the appli-

cations, and model transformation languages required for generating other models or code. Service-oriented design is a new domain which currently lacks effective and comprehensive domain-specific modeling languages and code generation tools.

In addition to the traditional class diagram model of the domain and the sequence and state diagrams modeling the behavior of objects and components, in the case of service-oriented software we need also to model the orchestration of services. Here we should consider such complications as compensation for actions in case of any failure during the process. Non-functional properties such as security, performance and reliable connection need to be modeled as well. They play a more relevant role in service-oriented computing than in traditional software. This is because services must respect service-level agreements which establish security policies and acceptance levels of performance. It is intuitive then to model the non-functional properties as contracts which are associated with the services, i.e. to follow a contract-based modeling approach.

3 Modeling Service-Oriented Systems

The UML [38] is the most well-known and mature language for modeling software systems. However, plain UML lacks native support for the specification of structural and behavioral aspects of services. Service modeling introduces a new set of key distinguishing concepts, for example partner services, message passing among requester and provider of services, long-running transactions, compensation, and events. Without specific support for those concepts in the modeling lan-

guage, diagrams quickly get overloaded with technical constructs, degrading their readability.

Several attempts have been made to add service functionality to the UML. Most notably, SoaML [40] is an upcoming standard UML profile of the OMG for structural specification of service-oriented architectures. Our own contribution to the field of UML service modeling is UML4SOA [34], a profile for specifying behavior of services, in particular service orchestrations, a feature which distinguishes service-oriented software from traditional application software. UML4SOA is based on the structural part of SoaML, adding the dynamic parts.

In the following sections we present our running example and give an overview of SoaML and UML4SOA concepts and discuss how to apply them to the case study scenario. We describe the *structure* of the service-oriented system and, finally, the *service orchestration behavior*. In each of these sections, we introduce the relevant SoaML and UML4SOA stereotypes used.

In later sections, the elements introduced here will be extended with a package of stereotypes for dealing with non-functional properties (Section 4).

3.1 The eUniversity Case Study

As a running example throughout this paper, we will consider modeling and implementing an all-electronic university (an eUniversity), in which all courses and paperwork are handled online. We will focus on the processing of a student appli-

cation for a course of studies. This example has been taken from one of the case studies of the Sensoria project [48, 1].

In this scenario, the student uses a website to apply for a certain course of studies. That is, the eUniversity website acts as a client to a service providing the functionality for handling a student application. This functionality is provided by an entity called the *ApplicationCreator*. Implementing this functionality requires the combination (orchestration) of a set of different external services, e.g. student office, a service for the upload of documents, and a service to check the application (validation service). This validation service, implemented by an entity called the *ApplicationValidator*, is itself also a composition of other services.

In the student application scenario of our eUniversity case study, the following non-functional requirements are defined:

- The *Client* and the *ApplicationCreator* should communicate via a secure and reliable connection.
- The document *UploadService* might be under heavy workload, therefore its throughput should be at least 10 requests/second with a 4 second average response time.
- All requests sent to the *ApplicationValidator* should be acknowledged.
- As the validation service handles confidential data, all requests should be encrypted in order to protect the privacy of the students.
- Messages sent by the *ApplicationValidator* must be clearly accountable, i.e. non-repudiation of messages must be guaranteed.

We will detail the model of the case study in the next two subsections and come back to the above requirements in later sections.

3.2 Modeling Structural Aspects

For modeling the structural aspects of our case study, we employ the basic UML mechanisms for modeling composite structures, enhanced with stereotypes from the SoaML profile - «*participant*», «*servicePoint*», «*requestPoint*», «*serviceInterface*» and «*messageType*» (listed in Table 1). With regard to the structure of our case study, we talk about services, service interfaces, and service participants. The basic unit for implementing service functionality is a service participant, modeled as a class with the stereotype «*participant*». A participant may provide or request services through ports, which are stereotyped with «*requestPoint*» or «*servicePoint*», respectively. Each port has a type, which is a «*serviceInterface*» implementing or using operations as defined in a standard UML interface definition.

The components of the eUniversity case study which are relevant for the student application scenario are shown in Fig. 2. It represents the overall composition of a SOA system modeled as a UML component diagram using SoaML model elements. As can be seen, each of our two participants offers or requires multiple services; for example, the *ApplicationCreator* is invoked by the client for the creation of a new application, but invokes several other services as well, such as the *validationService* and the *statusService*.

The eUniversity case study scenario includes two services which are defined as an orchestration of other services, the *ApplicationCreator* and the *Application-*

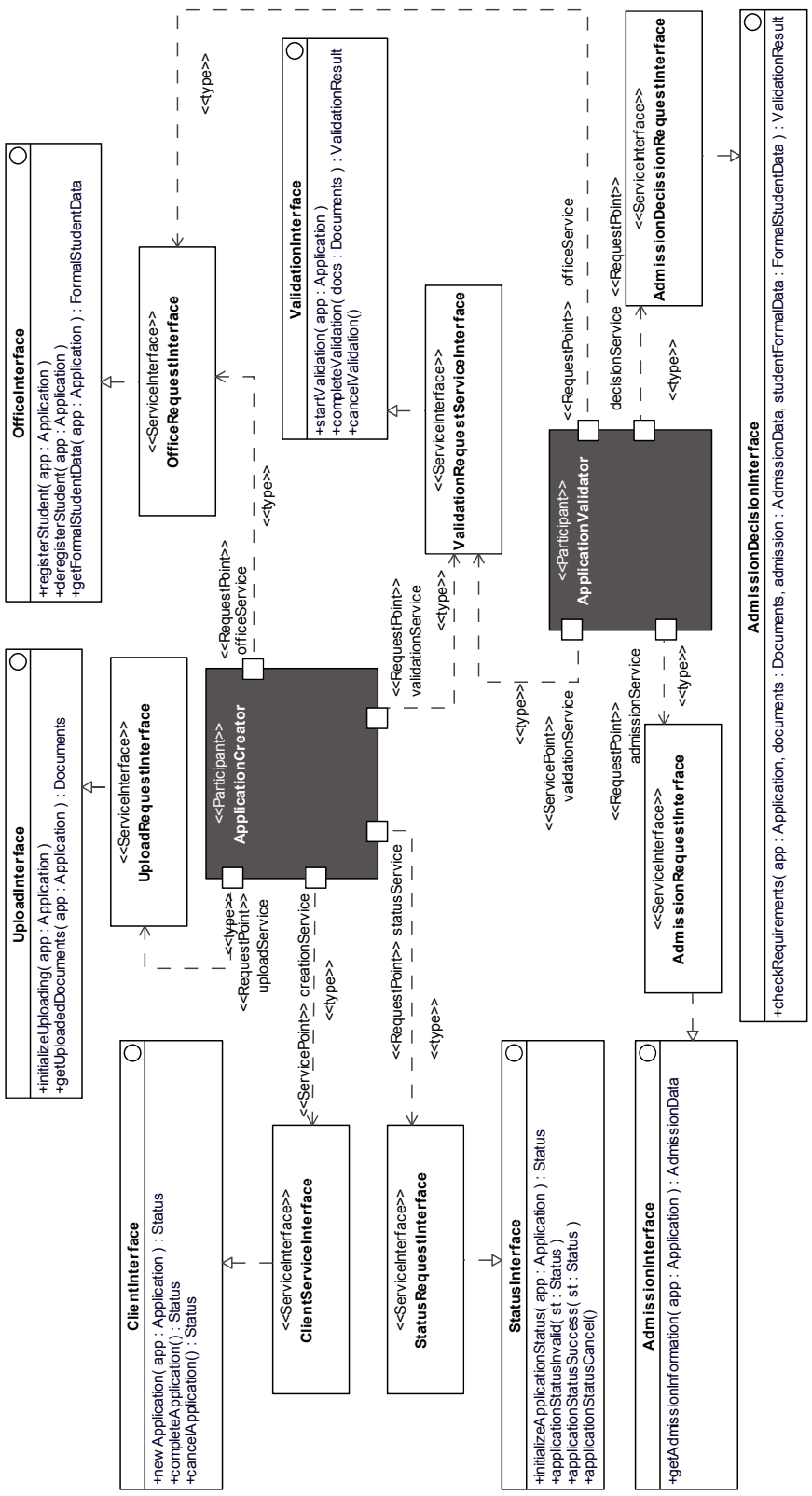

Fig. 2 The eUniversity Student Application Scenario

UML4SOA Metaclass	**Stereotype**	**UML Metaclass**	**Description**
Participant	«*participant*»	Class	Represents some (possibly concrete) entity or component that provides and/or consumes services
ServicePoint	«*servicePoint*»	Port	Is the offer of a service by one participant to others using well defined terms, conditions and interfaces. It defines the connection point through which a participant provides a service to clients
RequestPoint	«*requestPoint*»	Port	Models the use of a service by a participant and defines the connection point through which a participant makes requests and uses or consumes services
ServiceInterface	«*serviceInterface*»	Class	Is the type of a «*servicePoint*» or «*requestPoint*», specifying provided and required operations
MessageType	«*messageType*»	DataType, Class	Is the specification of information exchanged between service requesters and providers

Table 1 SoaML Profile

Validator (see colored components in Fig. 2). The behavior of each of these is modeled as an activity diagram which uses UML4SOA extensions. The objective of the *ApplicationValidator* is to verify whether the application follows the policies of the university. The actual implementation of the two orchestrations further refines the behavior of this scenario, and is detailed in Section 3.3. The other services, including the client service, are atomic and implemented in a standard programming language (for example, in Java).

Overall, the scenario works as follows: A student uses the website to apply for a certain course of studies. The website (not shown) contacts the *ApplicationCreator* through its *creationService* service port. The *ApplicationCreator*, in turn, calls other entities through the *uploadService*, the *officeService*, and the *statusService* ports. Last but not least, it also contacts the *ApplicationValidator* through the

validationService port for checking the student data and setting the status of the application. Being implemented as an orchestration itself, *ApplicationCreator* works with other entities itself – through the *officeService* (again), the *admissionService*, and finally the *decisionService* ports to carry out the validation task. After a review of the application by the various services, the student is notified whether he was accepted at the university.

Summarizing, services are defined as ports. Depending on whether they are provided or required, the stereotypes «*servicePoint*» or «*requestPoint*» are used. Ports belong to «*participant*»s, which may require or provide multiple services.

3.3 Modeling Behavioral Aspects

More challenging than modelling the structural aspects of SOAs is the task of modeling behaviour – in particular the orchestration of services. To enable developers to model such behaviour in an easy fashion, we have introduced UML4SOA [34], which is defined as a high-level domain-specific modeling language (DSML) for modeling service orchestrations as extensions of UML activity diagrams.

An excerpt of the UML4SOA metamodel is shown in Fig. 3, which includes the main concepts of our DSML and the relationships among these concepts. For each non-abstract class of this metamodel, we have defined a stereotype with the objective of producing semantically enriched but still readable models of service-oriented systems. Tables 2 and 3 provide a summary of the elements of the metamodel, the stereotypes that are defined for these metamodel elements (they comprise the profile UML4SOA), the UML metaclasses they extend, and a brief de-

scription. For further details on UML4SOA, including the full metamodel, the reader is referred to [34].

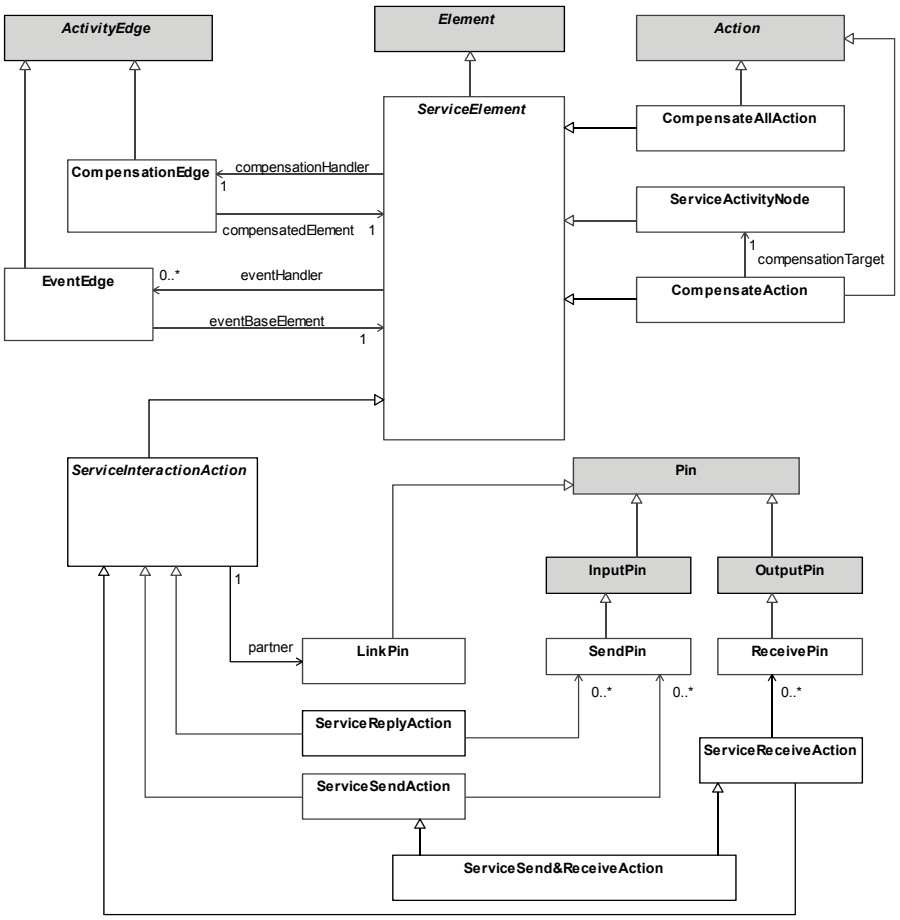

Fig. 3 Excerpt of the UML4SOA Metamodel (includes some colored UML metaclasses)

UML4SOA proposes the use of UML activity diagrams for modeling service behavior, in particular for modeling orchestrations which coordinate other services. We assume that business modelers are most familiar with this kind of notation to show dynamic behavior of business workflows.

UML4SOA Metaclass	**Stereotype**	**UML Metaclass**	**Description**
ServiceActivity Node	«*serviceActivity*»	Activity, Structured ActivityNode	Represents a special activity for service behavior or a grouping element for service-related actions
ServiceSendAction	«*send*»	CallOperationAction	Is an action that invokes an operation of a target service asynchronously, i.e. without waiting for a reply. The argument values are data to be transmitted as parameters of the operation call. There is no return value
ServiceReceiveAction	«*receive*»	AcceptCallAction	Is an action representing the receipt of an operation call from an external partner. No answer is given to the external partner
ServiceSend&Receive	«*send&receive*»	CallOperationAction	Is a shorthand for a sequential order of send and receive actions
ServiceReplyAction	«*reply*»	ReplyAction	Is an action that accepts a return value and a value containing return information produced by a previous ServiceReceiveAction action

Table 2 UML4SOA Profile (1)

The two processes *ApplicationCreator* and *ApplicationValidator* from Fig. 2 are modeled as UML4SOA orchestrations. The first one is shown in Fig. 4. It illustrates how the creator interacts with its partner entities through ports. It starts with a receipt («*receive*») of the call *newApplication* through the *creationService* service port, receiving the application. After the receipt of this call, *statusService* and *uploadService* are initialized (both times with synchronous calls, i.e. using «*send&receive*»), and the inital call is returned with a «*reply*». Completing the initalization phase, the *startValidation* call is sent (using an asynchronous «*send*») to the *ApplicationValidator* to request the start of the validation. After having done so, the process waits for another call («*receive*») from the client. The student will either press the button to complete the application, or another one to cancel it.

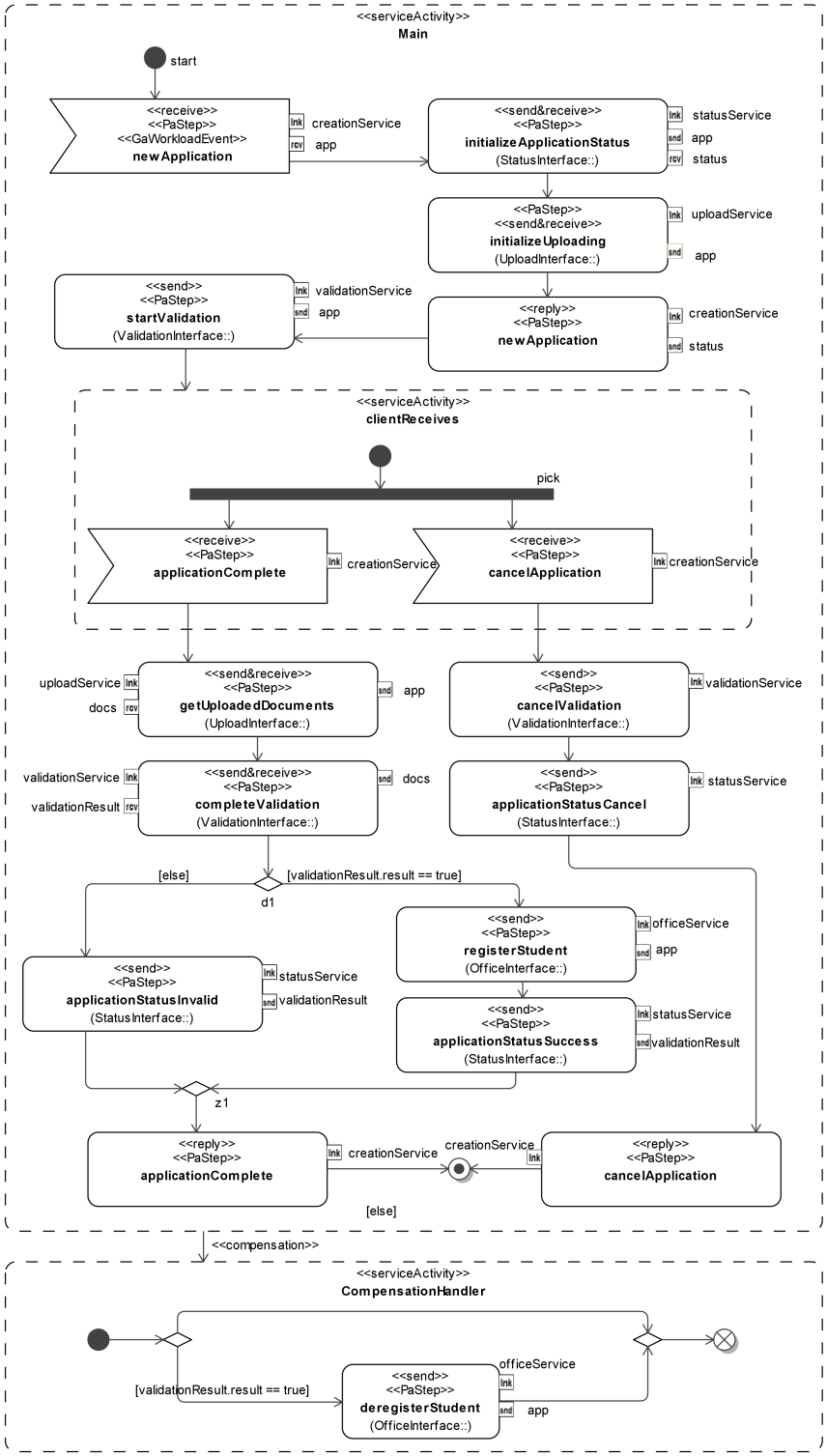

Fig. 4 UML4SOA activity diagram showing the ApplicationCreator

UML4SOA Metaclass	**Stereotype**	**UML Metaclass**	**Description**
CompensationEdge	«*compensation*»	ActivityEdge	Is an edge which connects an orchestration element to be compensated with the one specifying a compensation. It is used to associate compensation handlers to activities and scopes
EventEdge	«*event*»	ActivityEdge	Is an edge connecting event handlers with an orchestration element during which the event may occur. An event handler must start with a *receive*, and runs in parallel and in the context of the attached scope
CompensateAction	«*compensate*»	Action	Triggers the execution of the compensation defined for a (defined) scope or activity. Can only be used in compensation or event handlers
CompensateAllAction	«*compensateAll*»	Action	Triggers compensation of the scope attached to the handler in which the action is invoked, and all subscopes in reverse order of their completion. Can only be used in compensation or event handlers
LinkPin	«*lnk*»	InputPin	Holds a reference to the partner service by indicating the corresponding service point or request point involved in the interaction
SendPin	«*snd*»	InputPin	Is used in send actions to denote the data to be sent to an external service
ReceivePin	«*rcv*»	OutputPin	Is used in receive actions to denote the data to be received from an external service

Table 3 UML4SOA Profile (2)

If a *cancelApplication* call is received, the validation service is instructed to cancel the validation («*send&receive*»), and the status service is notified that the application has been canceled («*send*»). If, on the other hand, the student chose to complete the application, the uploaded documents are retrieved from the *uploadService* with a synchronous «*send&receive*» and a final validation is requested from the *ApplicationValidator*, using the *completeValidation* call (synchronous, «*send&receive*»). If the result is okay, the student is registered at the *studentOf-*

fice with *registerStudent* («*send*»). In any case, the initial call is replied to with a «*reply*» action.

Besides the normal flow of the activity, the diagram also shows a second structured activity node – a compensation handler. The actions defined within *CompensationHandler* are executed if the main activity has been completed successfully, but needs to be undone. This functionality can be triggered externally after the orchestration has been completed. If the application has been completed successfully before, the student is removed from the list of applicants by using a *deregisterStudent* call on the *officeService*.

Note that the activity diagram in Fig. 4 makes use of two distinct sets of stereotypes. The first set of stereotypes is part of the UML4SOA profile as defined above – i.e., the «*send*», the «*receive*», the «*send&receive*» and the «*reply*» stereotypes. The second set of stereotypes in Fig. 4 is part of the OMG MARTE profile, for example the «*PaStep*» or the «*GaWorkloadEvent*» stereotypes. Those are used for performance evaluation and will be discussed in the next section.

The second activity diagram, modeling the *ApplicationValidator*, is shown in Fig. 5. This service acts as supplier to the creation service, starting with the receipt of the *startValidation* call («*receive*») from the *ApplicationCreator* through the *validationService* port. Afterwards, both the *officeService* and the *admissionService* are contacted simultaneously and synchronously («*send&receive*») to check admission of the student, and to check the student data.

Subsequently, the process waits (using «*receive*») for the *completeValidation* call from *ApplicationCreator*. After it is received, all the information gathered so

far is checked with the help of the *decisionService*, and the result is returned via «*reply*» to the *ApplicationCreator*.

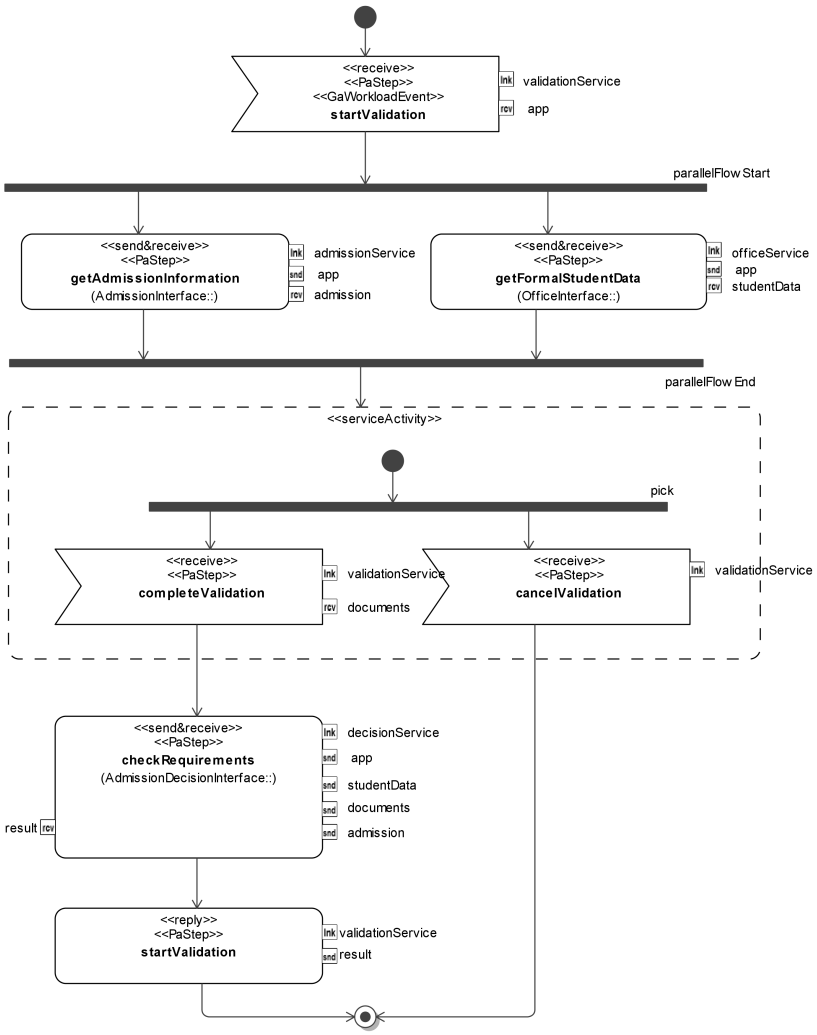

Fig. 5 UML4SOA activity diagram showing the ApplicationValidator

4 Enhanced Modeling with Non-functional Properties

The previous section has shown how the UML4SOA profile may be used to model (functional) static and dynamic aspects of a service-oriented system. In this section we extend the UML4SOA modeling approach for non-functional properties. We call the extension UML4SOA-NFP. First, we discuss non-functional aspects and standards of service-oriented systems, then we present the modeling elements for the structural and behavioral aspects and the corresponding stereotypes. Finally, non-functional properties of the case study introduced in Section 2 are modeled.

4.1 Non-Functional Aspects of Services

Performance characteristics describe the timely behavior of a service, such as response time, throughput, etc. Typically average and maximum/minimum values of these parameters are defined in service-level agreements (SLAs). In this paper, we present an analysis method on service level performance, however, middleware characteristics (e.g. maximum transmission time) can also be considered.

Dependability characteristics describe the behavior of the system in the presence of faults. Availability refers to the readiness of the service to be used while *reliability* of a service prescribes the capability of maintaining service quality (i.e., correct operation) [7].

Dependability can be defined at different levels in SOSs. Application level dependability describes requirements on the component behavior while middleware level dependability is related to the (Web) service layer and the message commu-

nication. The latter also hides the network level properties which can often change and typically are out of a service engineer's scope. UML4SOA-NFP can model both levels, however, the analysis and deployment methods presented here target the middleware level.

Reliable messaging in the field of traditional distributed systems is closely related to the guaranteed semantics of message delivery. Typical delivery modes are the following:

- *At least once delivery.* In the case of normal operation, every message is transferred at least once, with the possibility of sending multiple instances of the same message. This can only be allowed in systems where this does not have an undesired side-effect.
- *At most once delivery* guarantees that no message will be sent multiple times to the receiver, but their successful transmission is not ensured.
- *Exactly once delivery* is the strongest delivery semantics, guaranteeing both the successful message delivery (usually acknowledgements are required for each message) and the filtering of duplicate messages.

The following low-level attributes are required for the configuration of reliable messaging (besides **messagingSemantics**, which selects the messaging mode as described earlier):

- **inactivityTimeout**: (integer, seconds), after this period of time if no acknowledgment message has arrived, the connection is closed;
- **exponentialBackoff**: (boolean), if it is set to true, time amounts between retransmissions follow an exponential distribution;

- **acknowledgementInterval**: (integer, seconds), amount of time which should elapse before sending an acknowledgement message;
- **retransmissionInterval**: (integer, seconds), after this time a request is resent by the client if no acknowledgement has arrived.

Security The notion of *security* covers properties related to confidentiality (no unauthorized subject can access the content of a message), integrity (the message content cannot be altered), non-repudiation (which refers to the accountability of the communicating parties) and privacy (the identity and personal data of a client is not revealed to non-authorized bodies). Concepts such as authentication (checking the identity of a client) and authorization (checking whether a client might invoke a certain operation) are also of concern here.

In service-oriented systems, security should be guaranteed between service endpoints, independently from network level properties. This can be achieved using secure web services middleware. Message security is based on digital signatures and encryption of messages; here we distinguish the message header and body, however, further (application-specific) separation of message parts is also possible.

The following security parameters are used as a basis for configuration generation for secure communication middleware:

- **encryptBody, encryptHeader, signBody, signHeader** describe whether a security method is applied on (parts of) messages between client and service respectively
- **signAlgorithm** and **encryptionAlgorithm** determine the security algorithms

– **authTokenType** determines the type of the security token (e.g. username or binary)
– **useTimestamp** allows the user to specify timestamps for messages

Note that the executable set of security configurations is restricted in current middleware to certain combinations of the above parameters, therefore we propose default values in the profile which conform to the actual deployment possibilities.

4.2 Extending Structural Models with Non-functional Properties

This section describes the UML extension for modeling non-functional parameters related to structural models of services. On the one hand, these models rely upon the General Resource Model (which is part of the UML Profile for Schedulability and Time [36]) and UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms [37]. However, the way UML4SOA-NFP handles these parameters also conforms to the service management of typical business applications using Service Level Agreements (SLA).

A metamodel for non-functional properties. Fig. 6 shows the metamodel of non-functional concepts and their relationships. For each additional concept we define a UML stereotype.

Since in real service configurations, service properties can vary for different classes of clients, we follow a contract-based approach, where non-functional properties of services are defined between two «*Participant*» components, namely, the service provider and the service requester. These contracts are modeled by «*NFContracts*».

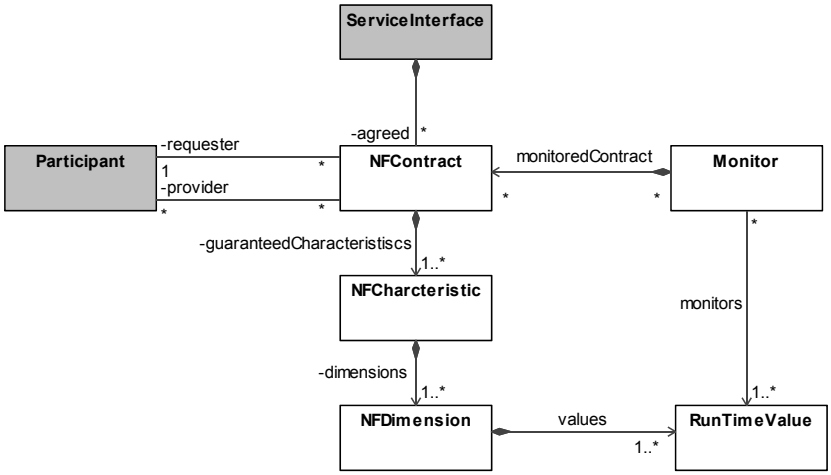

Fig. 6 Metamodel of non-functional extension

Different non-functional aspects (performance, security, etc.) are modelled in corresponding «*NFCharacteristics*» which group different properties (e.g., response time) in «*NFDimensions*» (where a «*RunTimeValue*» is associated to each dimension). The reason for creating separate classes for measureable values instead of actually storing them in attributes is to correlate real SLAs where most parameters are typically bound to a range of allowed values. Moreover, concepts like average values, deviation, etc. need to be modeled in a uniform way.

During a negotiation process, participants create an *agreed* contract of their *provided* and *requested* contract specifications.

Finally, properties of services need to be monitored at runtime (modeled as «*Monitor*») either by the participating parties or by involving a separate entity.

Modeling non-functional properties in UML4SOA-NFP. On the UML class level, each contract is modelled using a UML class with the stereotype «*NFContract*». Each characteristic (tagged by «*NFCharacteristic*») is another UML class asso-

UML4SOA-NFP Metaclass	**Stereotype**	**UML Metaclass**	**Description**
NFContract	«*NFContract*»	Class	Represents a non-functional contract between a service provider and a service requester
NFCharacteristic	«*NFCharacteristic*»	Class	Represents a non-functional aspect such as performance, security, reliable messaging, etc.
NFDimension	«*NFDimension*»	Class	Groups non-functional properties within a non-functional aspect (characteristics)
RunTimeValue	«*RunTimeValue*»	Attribute	An actual non-functional property
Monitor	«*Monitor*»	Class	A run-time service to monitor a contract (not used in the paper)

Table 4 UML4SOA-NFP Profile

ciated to the respective contract. Each dimension is also defined by a UML class stereotyped as «*NFDimension*». The actual runtime values of each dimension are defined as UML properties. Stereotype usage is summarized in Table 4.

The actual non-functional parameters within a contract are set by using an object diagram instantiating these classes (and attributes).

Modeling non-functional properties of the eUniversity. Figure 7 illustrates how the non-functional requirements presented in Section 2 are captured in a UML4SOA model by defining a *non-functional contract* between *ApplicationCreator* and *ApplicationValidator*.

The requirements of Sec. 2 are mapped to three «*NFCharacteristics*», namely Performance, Reliable Messaging and Security (as discussed in Sec. 4.1).

– *Performance* aspects include two «*NFDimension*» elements, namely response time and throughput. Throughput definition consists of defining a guaranteed throughput and a maximal throughput, while for response time, the contract contains an average value and a maximum value.

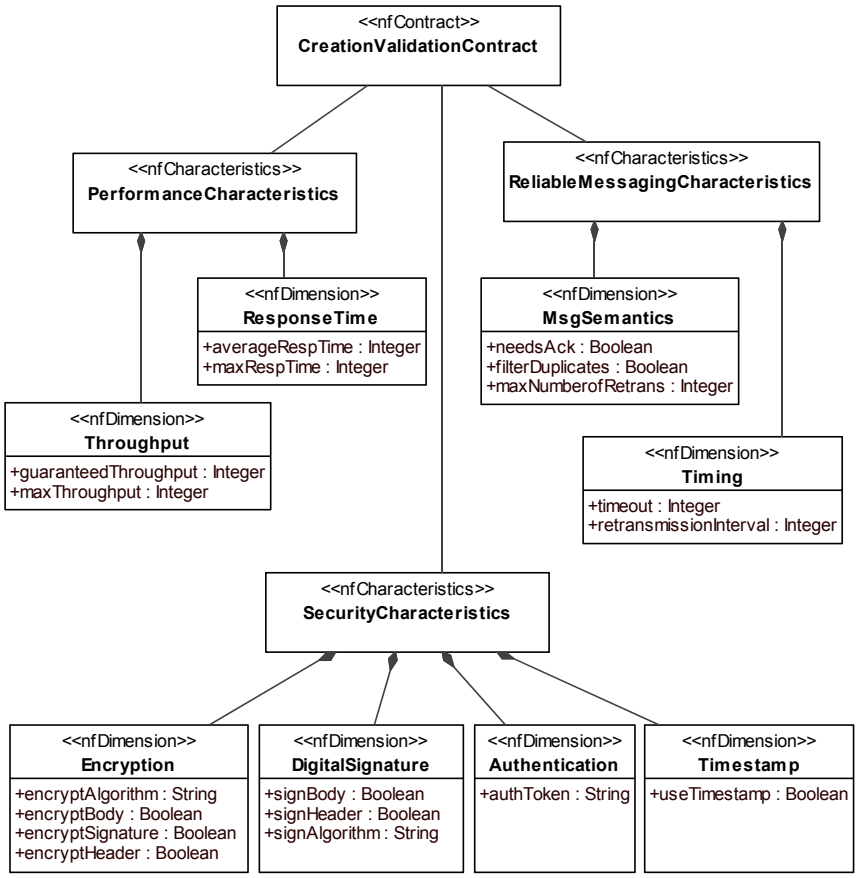

Fig. 7 Elements of the contract between ApplicationCreation and ApplicationValidation services

– *Reliable messaging* parameters first contain the required message semantics by stating whether acknowledgement is required or duplicate messages are allowed. Timing dimensions include the timeout for considering a message lost and the retransmission interval.

– *Security* aspects are composed of dimensions encryption, digital signature, timestamp and authentication. In our example, the latter is simplified to contain only the authentication token type.

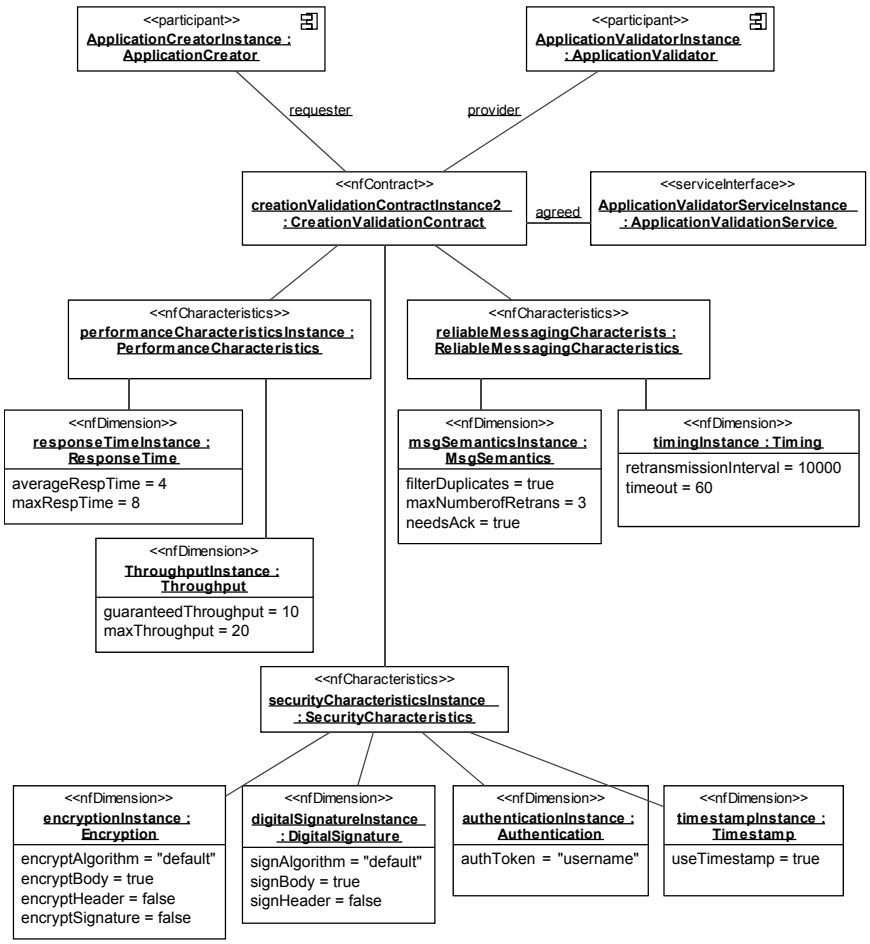

Fig. 8 Instance model with non-functional properties

A concrete non-functional service configuration (on the object level) is shown in Fig. 8. This instantiates Fig. 7, and assigns concrete values to the run-time values of non-functional parameters.

For instance, *MsgSemanticsInstance* prescribes that each message between the two orchestrators needs an acknowledgement and the system can resend each message at most three times. Moreover, duplicate messages also need to be filtered. On the security level, timestamps are required to be used (*TimestampInstance*), while

users are authenticated by their username (*AuthenticationInstance*). Later, these reliable messaging and security specifications will be used by deployment transformations of Sec. 6.

4.3 Extending Behavior Models with Non-functional Properties

We make use of the MARTE profile for to annotate UML models with non-functional properties required for performance evaluation. In addition to being useful for documentation purposes, these models will be subject to automatic extraction of quantitative estimates.

Performance models offer insights into the dynamic understanding of complex service-oriented systems which are complementary to those which can be obtained through measurement and profiling. Measurement allows us to understand the system as it is today: modeling allows us to understand how it could be tomorrow. Predictive performance modeling considers alternative designs or improvements, and evaluates these to identify the adaptation of the system which will give the greatest improvement with respect to a given performance goal (such as reducing response time). Measurement and modeling are intimately linked because accurate measurement provides the parameter data which models need in order to make valuable predictions.

In order to build a coherent performance model for performance analysis we must describe the *workload* placed on the system and the cost of the individual units of execution (*activities*) which make up the events of the model. Performance evaluation may be carried out on activities stereotyped with «*GaScenario*». Its

cause property allows the extraction of workload specification, stereotyped with «*GaWorkloadEvent*». Closed patterns are supported, which define the workload as a population of users which interpose some thinking time between successive, cyclic executions of the activity. Workloads (defined by «*GaWorkloadEvent*») in the following form are accepted:

```
pattern = closed(population=M,extDelay=(exp(1/r),s))
```

which indicates a closed workload of M users which cyclically execute the activity. An exponentially distributed thinking time with mean duration `1/r` seconds is interposed between successive requests.

The atomic units of execution are stereotyped with «*PaStep*». To denote the amount of time taken by a step we use its *hostDemand* attribute. Meaningful applications will typically have `hostDemand = (exp(<time>), s)` to indicate an exponentially distributed delay with mean `<time>` seconds. The execution rate of an action will be extracted from the «*PaStep*» application.

The use of these stereotypes can be observed in the activity diagrams of *ApplicationCreator* (Fig. 4) and *ApplicationValidator* (Fig. 5).

5 Early Estimation and Evaluation of Non-Functional Properties

For the quantitative analysis of non-functional service attributes, the timed process algebra PEPA can be employed as the intermediate formalism derived from UML models of services annotated with UML4SOA and MARTE. The current section provides a brief (and high-level) overview of how formal performance models are derived from service models with a special focus on insights gained by analysis

specific to our case study. For a detailed presentation of the transformation, the reader is referred to [43].

5.1 Overview of PEPA

5.1.1 Language elements. PEPA is a formal language which allows the definition of models as a composition of interacting automata (*sequential components*). Sequential components may carry out activities independently of the rest of the system, or in cooperation (i.e., synchronization) with other automata. The operators supported by the language are informally introduced below. For a complete formal definition the reader is referred to [25].

Prefix $(\alpha, r).P$ denotes a process which performs an action of type α and behaves as P subsequently.

Choice $P + Q$ specifies a component which behaves either as P or as Q. The activities of both operands are enabled and the choice will (stochastically) behave as the component which first completes.

Constant $A \stackrel{def}{=} P$ is used for recursion. Cyclic definitions are central in the characterisation of the underlying continuous-time Markov chain derived from a PEPA model.

Cooperation $P \underset{L}{\bowtie} Q$ is the compositional operator of PEPA. Components P and Q synchronize over the set of action types in set L; other actions are performed independently. For example, $(\alpha, r_1).(\beta, s).P \underset{\{\alpha\}}{\bowtie} (\alpha, r_2).(\gamma, t).Q$ is a composition of two processes which execute α cooperatively. Then, they perform action β and γ independently and behave as P and Q, respectively.

The operator $\|$ is sometimes used as shorthand notation for a cooperation over an empty set, i.e., $\underset{\emptyset}{\bowtie}$. Independent copies of a component are indicated by the notation $P[N] \equiv \underbrace{P \parallel P \parallel \cdots \parallel P}_{N}$

5.1.2 Rates of activities. An activity is associated with an exponential distribution with mean duration $1/r$ time units. Generally distributed activities can be obtained by using suitable phase-type distributions, although these will not be discussed further in this paper. The symbol $\top$ specifies a *passive* rate which may be used to model unbounded capacity. The duration of an activity involving passive rates is determined by the active rate of the synchronizing components.

Cooperating components need not have a common view of the duration of shared actions. The semantics of PEPA specifies that the rate of a shared action is the slowest of the individual rates of the synchronizing components, e.g., $\min(r_1, r_2)$ in the example above.

In order to carry out a quantitative analysis, PEPA models are interpreted as continuous-time Markov chains. In particular, in Sec. 5.5, we will give examples of analysis of the long-run behaviour of a system (*steady-state analysis*).

5.2 From UML Activity Models to PEPA

5.2.1 Overview of System Equation and Workload The transformation from service models captured using UML4SOA and MARTE profiles gives rise to a system equation of the target PEPA model in the following form:

$$System \stackrel{def}{=} Workload \underset{\{\alpha\}}{\bowtie} \mathcal{A}[K]$$

Here $Workload$ represents the (abstract) behavior of M independent users of an activity. An individual workload component is modeled as a two-state automaton

$$\begin{aligned} Think &\stackrel{def}{=} (think, r).Start \\ Start &\stackrel{def}{=} (\alpha, \top).Think \end{aligned} \tag{1}$$

where the passive activity $(\alpha, \top)$ captures the fact that the action type α represents the first unit of computation performed by the system, and the rate is determined by the other synchronizing components. Thus, the overall PEPA sub-system for an array of M independent users is:

$$Workload \stackrel{def}{=} Think[M]. \tag{2}$$

The notation $\mathcal{A}$[K] represents an array of concurrent flows derived by a model transformation from UML activity diagrams of service models, which is discussed below.

5.2.2 Basic Transformation Blocks For space considerations, we only present a high-level overview of the main blocks of the transformation (Fig. 9), and interested readers are referred to [43] for further details.

5.2.3 PEPA model of the running example. The translation algorithm can be applied to the activity diagrams in Figures 4 and 5. The corresponding sub-systems of the performance model are shown in Figures 10 and 11, respectively.

The model of *ApplicationCreator*, denoted by $ACS::StartCreation$, consists of a single sequential component with two choices $ACS::Pick$ and

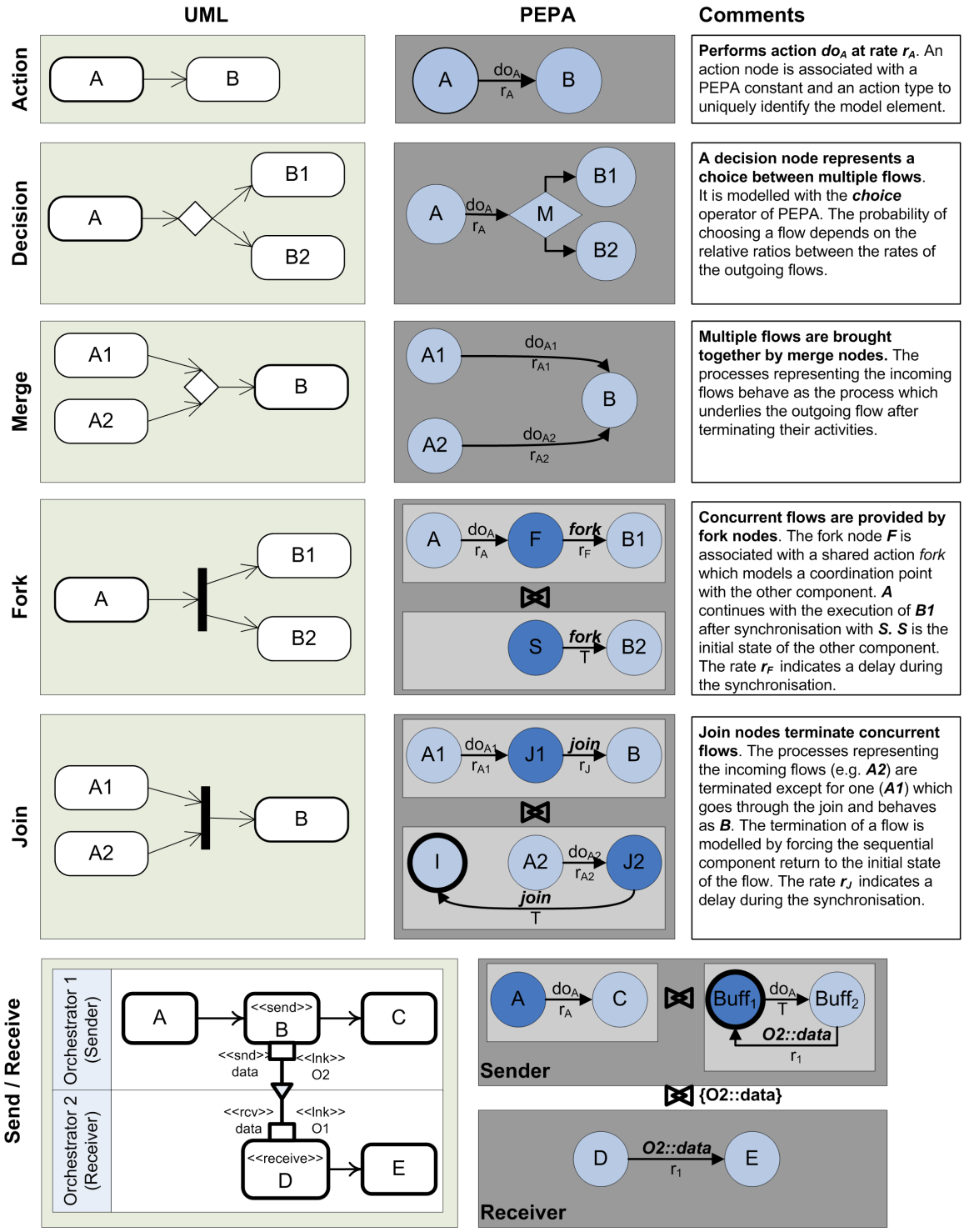

Fig. 9 Overview of UML-to-PEPA transformation

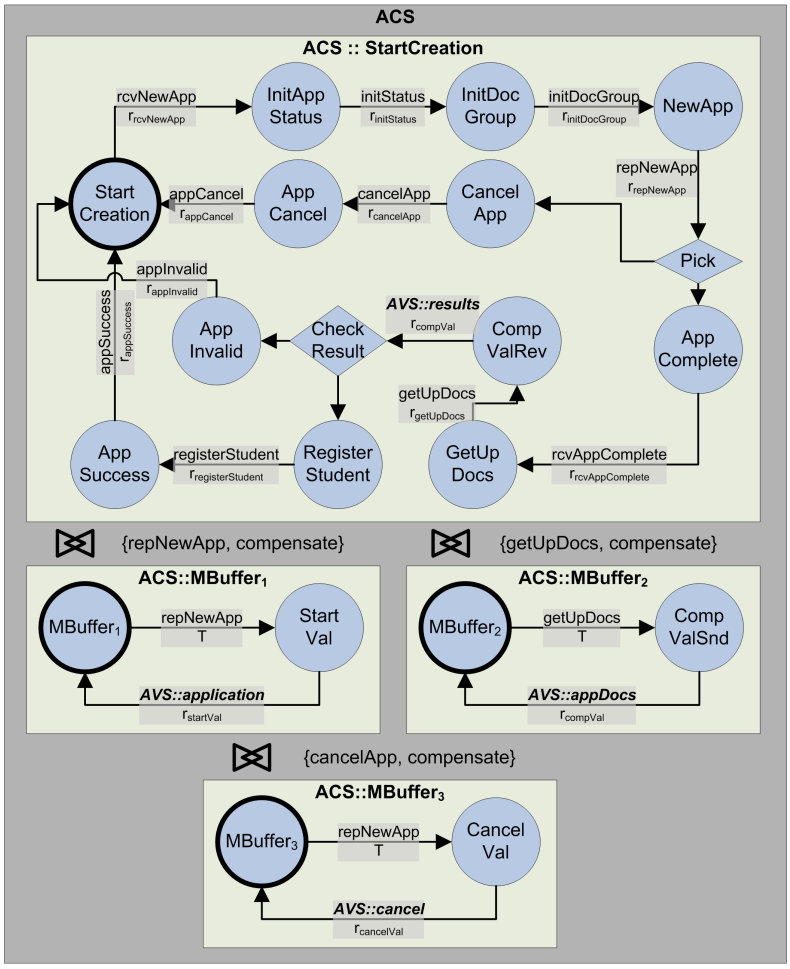

Fig. 10 PEPA model of *ApplicationCreator*

$ACS::CheckResult$, corresponding to the decision nodes *pick* and *d1*, respectively. The model of *ApplicationValidator*, denoted by $AVS::StartVal$, has two concurrent flows of execution. The second flow performs the action *checkData* and synchronize with the first flow at nodes *parallelFlowStart* and *parallelFlowEnd*. Thus, the second flow is mapped onto the three-state sequential component evolv-

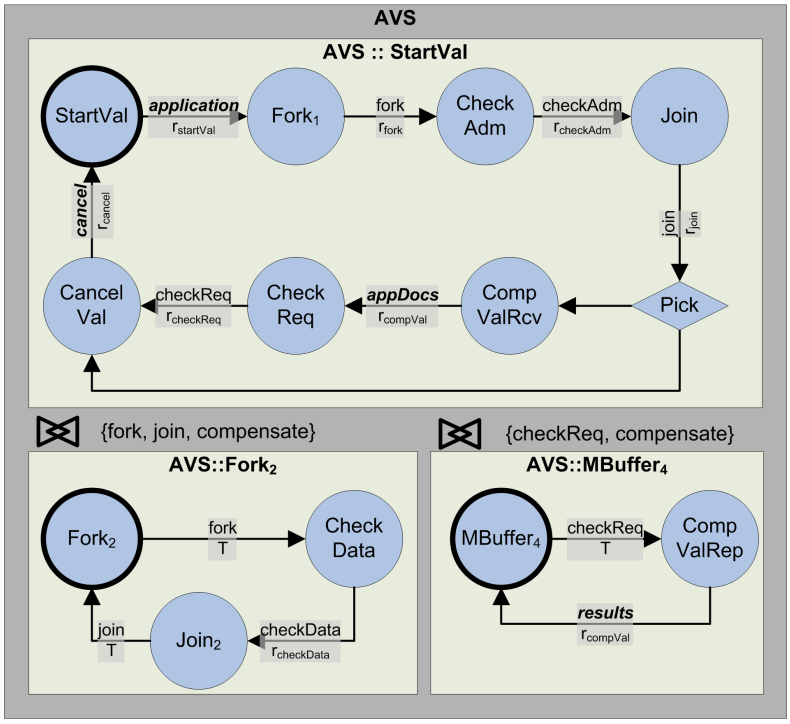

Fig. 11 PEPA model of *ApplicationValidator*

ing through local states $AVS::Fork_2$, $AVS::CheckData$, and $AVS::Join_2$. Additional (message buffer) components will be discussed in the sequel.

5.3 Handling Interaction between Orchestrators

Now we aim at capturing the interplay between the two orchestrations as specified by the stereotype applications of UML4SOA. For this purpose, we model the transmission of a message between two orchestrators using message buffers. Cooperation is modeled by exploiting the compositionality of the PEPA language as the core building blocks (discussed in Sec. 5.2) are gradually extended by message buffer components.

In other words, our aim is to extract a PEPA model in the form:

$$System \stackrel{def}{=} Workload \underset{\mathcal{L}}{\bowtie} \Big(ACS[K_{ACS}] \underset{\mathcal{M}_B}{\bowtie} B[S_B] \underset{\mathcal{M}_C}{\bowtie} C[S_C] \cdots \Big)$$

$$\underset{\mathcal{M}}{\bowtie} \Big(AVS[K_{AVS}] \underset{\mathcal{M}_D}{\bowtie} D[S_D] \underset{\mathcal{M}_E}{\bowtie} E[S_E] \cdots \Big)$$

where $\mathcal{L}$ contains the initial action executed in the scenario and the cooperation set $\mathcal{M}$ has the action types which correspond to the exchange of messages between the two orchestrators. The core building blocks of a component are extended with components $B, C, \ldots$ and $D, E, \ldots$, which model message buffers for asynchronous communication between the two orchestrators ACS and AVS. The sizes of the message buffers, i.e., $S_B, S_C, S_D, S_E, \ldots$ are extracted from MARTE annotations. The cooperation sets $\mathcal{M}_B, \mathcal{M}_C, \mathcal{M}_D, \mathcal{M}_E, \ldots$ contain the activities which cause an asynchronous delay to be sent.

5.3.1 Extraction of Message Buffers The components for message buffers are extracted from a Composite Structure diagram such as that in Fig. 2.

The transformation of an action node takes account of the input and output pins as well as the UML4SOA stereotype application to the node itself: «*send*», «*receive*», «*send&receive*», or «*reply*».

If the «*lnk*» pin references an element which is not an orchestration, it is handled as an action node. That is, although the UML4SOA profile indicates communication with other participants the exchange is abstracted away with an atomic activity in the performance model, because the concrete behaviour of the link is not available. Conversely, if «*lnk*» references an orchestrator O_i, then the message

exchange is modeled as a shared action between the two activities. The set of such shared action types is called the *interface* of an activity, denoted by $\mathcal{I}_i$. Interfaces will be used during the generation of the overall system equation. In the following, we only describe the treatment of a pair of «*send*» and «*receive*» action nodes in detail.

«send» node Let O_1 be the orchestrator which has a «*send*» node. The synchronizing orchestrator, O_2, can be retrieved by the reference *node.lnk*. According to the semantics of UML4SOA introduced in Section 3.3, the matching node of O_2 must be stereotyped with either «*receive*» or with «*receive&send*». The algorithm also requires that the reference of «*rcv*» in the receiving node be equal to the reference of «*snd*» in the sending node. Thus a shared action type may be constructed by inspecting *node.lnk* and *node.snd*. This shared action will be added to $\mathcal{I}_1$. The rate extracted from the application of «*PaStep*» indicates the local rate of the shared activity. Notice that the translation of the action node does not require the traversal of the cooperating orchestrator's activity. The pattern of transformation is shown in Fig. 9.

The UML4SOA profiles states that «*send*» indicates asynchronous communication. In PEPA, this is captured by associating a message buffer of finite size with each «*send*» node, and each place in the buffer is modeled as a two-state sequential component. The first state (i.e., $Buff_1$) of the component observes the execution of the action that precedes the asynchronous send. Observation is modeled as a passive cooperation between a sender's flow and a buffer place. The second state (i.e., $Buff_2$) models the transmission to the remote orchestrator. The non-blocking

behavior of the sender's flow is expressed by the fact that the flow is not involved in the transmission of the message—it behaves as the process which follows the «*send*» node after the preceding activity is completed.

«receive» node A «*receive*» is blocking, hence the shared action denoting the communication with the remote orchestrator is performed by the receiving flow. (see bottom component of Fig. 9). In this case the shared action type is constructed by traversing the pins stereotyped with «*rcv*» and «*lnk*».

5.3.2 Communication between Orchestrators The communication of *ApplicationCreator* (see bottom part of Fig. 10) with the orchestrator *ApplicationValidator* (Fig. 11) is handled by three message buffers, i.e., $ACS::MBuffer_1$, $ACS::MBuffer_2$, and $ACS::MBuffer_3$. The first state of the buffer observes the execution of one action of the main flow, and the second state performs the transmission of the message. The shared action types are named by using the format `lnk :: snd` (similarly, `lnk :: rcv` is used for action nodes stereotyped with «*receive*»). Components $ACS::CompValSnd$ and $ACS::CompValRcv$ model the two-phase PEPA behavior of the node *compVal*, stereotyped with «*send&receive*». Notice that cooperation occurs over distinct action types $AVS::appDocs$ (send) and $AVS::results$ (receive). The matching underlying sequential components of *ApplicationValidator* are $AVS::CompValRcv$ and $AVS::CompValRep$, between which the independent action $checkReq$ is performed.

The overall system equation is

$$System \stackrel{def}{=} Workload \underset{\{rcvNewApp\}}{\bowtie} \Big(ACS \underset{\mathcal{M}}{\bowtie} AVS \underset{\{compensate\}}{\bowtie} Compensator \Big) \quad (3)$$

where $\mathcal{M} = \mathcal{I}(ACS) \cup \mathcal{I}(AVS) \cup \{compensate\}$, and

$$\mathcal{I}(ACS) = \mathcal{I}(AVS) = \big\{ AVS::application, \\ AVS::appDocs, AVS::results, AVS::cancel \big\}.$$

5.4 Compensation and Exception Handling

Compensation and exception handling represent reactions to adverse situations during the course of an orchestration. From a performance standpoint, these events can be treated similarly—the current flow of control halts and passes on to some handler which performs a series of activities to restore the system. The performance model introduces failure in the orchestrations as activities competing with the business logic activities. Failure activities are represented by a choice operator which is added to all the local states of the PEPA sub-systems underlying the orchestrations.

When a failure occurs, the business logic flows of all the orchestrators are reset to their initial conditions (by synchronization of the flows over the failure action type) and the control is passed on to a sequential component which models the handler, according to the behavior described in the handling scope. The failure rate is attached as a MARTE annotation to the edge which triggers the handler.

For our running example, the compensator sequential component (Fig. 12) is triggered by the execution of the $compensate$ action and is defined in Fig. 12.

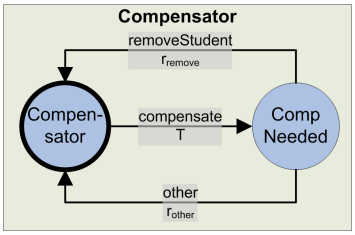

Fig. 12 PEPA model of *Compensator*

5.5 Performance Evaluation of the Case Study

To gain insight into the behavior of a system a common practice is to carry out *sensitivity analysis*, which studies the impact that certain parameters have on the overall performance. In this section, the performance metric of interest will be steady-state *throughput*, which gives the frequency at which an activity is performed in the system at equilibrium. As with most performance studies, throughput analysis is a useful approach because it summarises effectively the dynamic behavior of the system, accounting for delays due to fork/join synchronisation mechanisms, message passing, and computation cost associated with each basic activity of the system.

5.5.1 Sensitivity analysis: Fixed rates, Varying workload. An interesting sensitivity analysis is concerned with establishing how varying workload intensities affect system-level non-functional parameters. For instance, in our case study a

suitable index to be measured is the throughput of the action $appSuccess$ in the underlying PEPA model.

The set-up for workload analysis consists in the solution of the model for increasing population levels of users, represented by the array $Think[M]$. The analysis is specified by using MARTE annotations in the UML model containing a root activity stereotyped as «*GaWorkloadEvent*»:

```
pattern =
closed(population=in:M,extDelay=(exp(1/r),s))
```

where `in:M` indicates an input variable for the performance model, which is bound to an integer before the model is analyzed. The performance metric is specified by setting the following property in the «*PaStep*» application of node *applicationStatusSuccess*:

```
throughput=out:appSuccessTh
```

Figure 13 shows a typical result for this form of analysis. Under low intensity, the throughput of the system increases with the number of users. This is the behavior observed for our model for $M < 93$. However, the system's concurrency levels cannot meet higher demands as the population is increased further. This degradation corresponds in the graph to a flat throughput for $93 \leq M \leq 100$.

5.5.2 Sensitivity analysis: Fixed workload, Varying rates. An orthogonal analysis approach may concern the *sensitivity of the system performance to a specific activity rate*. Here all the other parameters of the system, including the workload

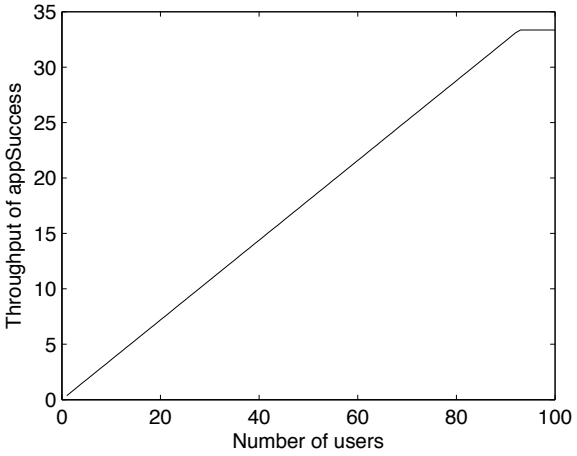

Fig. 13 Workload analysis studies how the user population affects performance of the system. Here, the performance metric of interest is the steady-state throughput of processing applications to e-University courses. Non-degrading performance is observed for population sizes less than 93.

specification, are fixed. The activity under study is varied across a range of suitable rate values and the corresponding performance measures are calculated. In our example, the activity node *checkProgramRequirements* in *ApplicationValidator* may play a crucial role. This activity is interposed between two nodes which represent communication with *ApplicationCreator*. Therefore *ApplicationCreator* is blocked during the course of the activity. Intuitively, one may conclude that increasing the activity rate corresponds to an increase in the system performance. Although this holds true, the relationship is not linear thus it is interesting to determine the range of values in which the relative gain is the highest.

Figure 14 shows the sensitivity analysis of $r_{checkReq}$ in the interval $[10, 200]$ with respect to the previously discussed system throughput. Indeed, the graph reveals that an optimal relative gain is obtained for values around 50 and further increases—for instance, doubling the rate from 100 to 200—yield smaller and

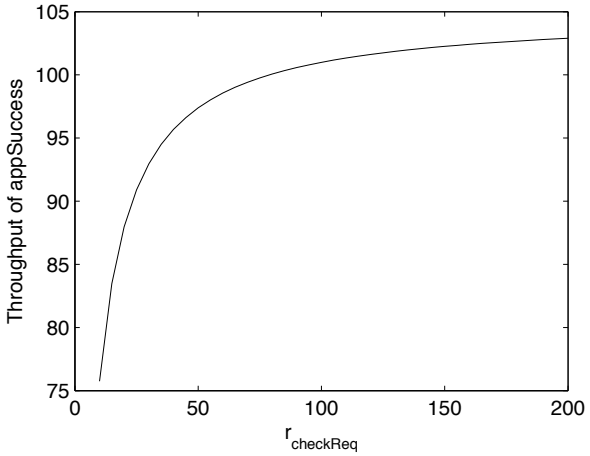

Fig. 14 Sensitivity analysis of $r_{checkReq}$.

smaller improvement. Similarly to the previous case, sensitivity analysis may be specified in the UML model by using the following property for the «*PaStep*» application to node *checkProgramRequirements*:

```
hostDemand=(exp(1/in:r_checkReq),s)
```

6 Automating Service Deployment by Model Transformations

Due to the rapid increase in the number of available services, greater emphasis is put on their non-functional aspects as described in Sec. 1. In order to meet such non-functional requirements, a service needs to be designed for reliability by making design decisions on an architectural level. However, this often conflicts with the current tool support for service development which has a relatively low level of functionality (merely creating appropriate XML descriptors, service configuration files, etc.)

Recently, the identification of non-functional parameters of services has been addressed by various XML-based standards related to web services. As web service communication assumes unreliable transfer by default, some standards (such as WS-ReliableMessaging [4] and WS-Reliability [3]) aim at ensuring reliable message communication. Security-specific configuration parameters are described in the WS-Security standard [2]. A brief summary of the contents of these standards were provided in Sec. 4.1 and 4.2.

Unfortunately, the manual creation of such service configuration files is typically an error-prone task during the deployment of services as XML parsers do not protect us against setting a syntactically correct but semantically incorrect value within a configuration file. Moreover, web services standards capture different subsets of non-functional parameters making even closely related standards incompatible with each other. Furthermore, unsurprisingly, each specific middleware implements the standard slightly differently. In addition to that, non-functional properties are captured at a low implementation-level by using dedicated XML deployment descriptors. As a consequence, (i) service configurations cannot be designed at a high architectural level, and (ii) the portability of service configurations is problematic. As the support of non-functional aspects in service platforms is changing rapidly, we propose an approach compliant with the Model-Driven Architecture (MDA) principles for the deployment of service configurations [28].

6.1 Target Deployment Languages and Transformation Flow

For this purpose, we created PIM2PSM and model-to-code transformations to facilitate service development for reliable and secure middleware. These transformations currently handle *reliable message communication* and *security* in service-oriented systems. Our transformation suite enables the automated generation of structural service descriptors and deployable policy files which determine the run-time behavior of services w.r.t. reliability and security requirements. Its modular implementation allows for future extension in other non-functional domains (e.g., logging) and other service platforms (e.g., SCA) as well. Standards-compliant non-functional service configurations make it necessary to synthesize one or more XML configuration files as deployment descriptors.

The actual model transformations can be realized through several steps. Below we exemplify one possible workflow for obtaining the models in the complex chains of model transformations.

- **PIM models**: The input of the chain is a standard UML2 model developed using EMF and serialized as XMI, which uses the UML4SOA(-NFP) Profile.
- **PSM models**: After the extraction of relevant model parts, internal service models are generated within the model transformation tool (describing core services «*SOA model*», reliable messaging setup «*SOA RM model*», security «*SOA security*», etc.). These are then processed in order to create descriptor models (e.g. «*WSDLmodel*», «*RAMP model*», «*Sandesha model*») which conform to industrial standards. These can be considered as PIM2PSM mappings in the MDA terminology.

- **Target XML files**: These descriptor models are the basis of XML file generation. These files are directly usable as configuration descriptors on standard platforms. Besides the server-side configuration XMLs (namely one for reliable messaging and one for security aspects), WSDL files of the services are also created. These are PSM2CODE transformations.
- **Glue code for deployment**: In case of the Apache Axis platform, deployable server-side projects are also created by Java applications. These have to be extended with the implementation (source files) of the services.

An overview of the core model transformation problem for deriving server-side configuration files of the Apache platform is presented with a description of transformation steps in Fig. 15. The trace model we use creates connections between source and target elements (objects) in the form of typed relations to ease transformation development. Moreover, such models also can be used to trace requirements from the high-level models to the code.

This transformation scheme is uniformly applicable to different «*NFCharacteristics*» (with minor adjustments to handle names of elements in case of «*$name*»). As a further technical detail, it is worth pointing out that certain default values can be set by the transformation itself (i.e. the source UML model does not need to contain them as in case of «*exponentialBackoff*»). Finally, certain configuration parameters in the model might not be required by the underlying middleware.

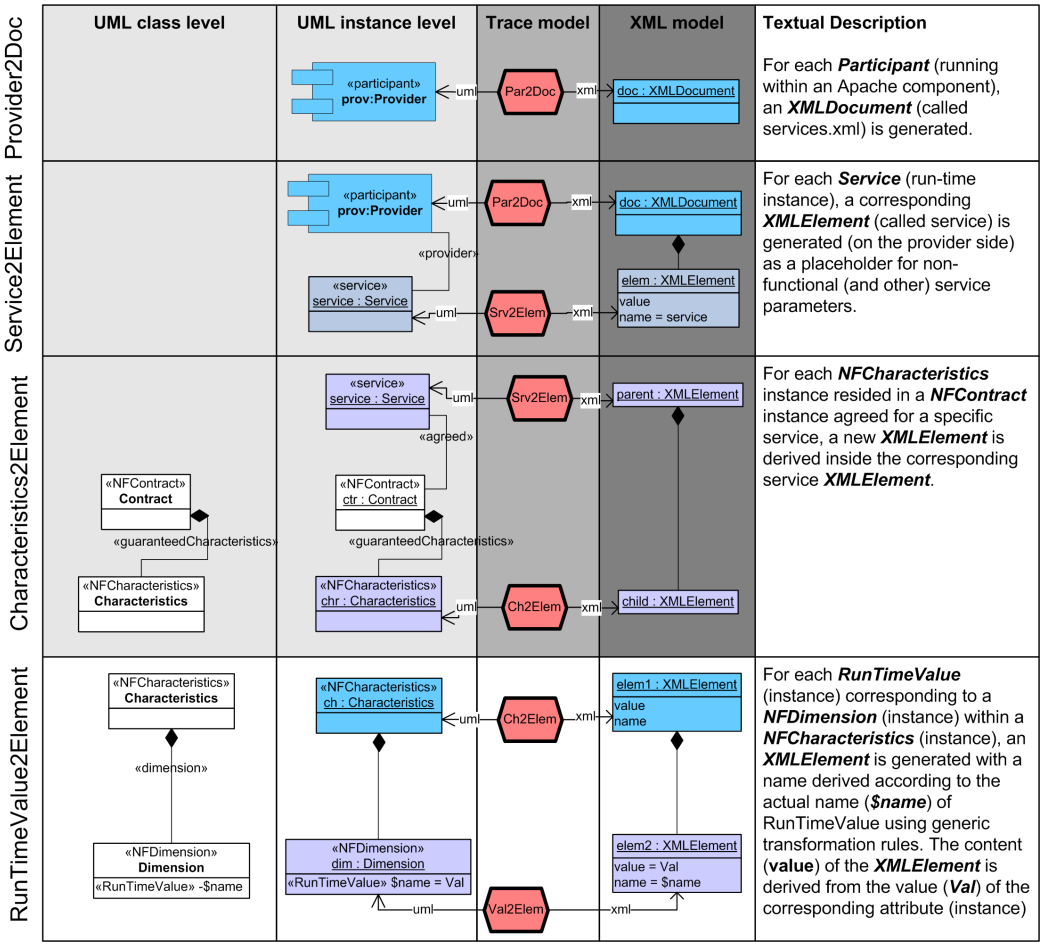

Fig. 15 Overview of deployment transformations

6.2 Transformation Implementation in VIATRA2

Transformations were implemented in the VIATRA2 framework [45] which is a modular, open source model transformation framework built on Eclipse, which supports the efficient design and execution of model transformations. Transformations are defined by graph transformation rules (i.e., declarative description of model patterns) and Abstract State Machines, which provide an intuitive yet

precise way of capturing complex transformations. The choice of the VIATRA2 framework can also be explained by the support for generic transformations [46], which significantly reduces the number of transformation rules.

The transformation implementation process for our deployment transformations consists of the following conceptual steps:

1. Create metamodels for source and target domain. An example is the domain of UML as source domain and a simple representation of services, connections and reliable messaging constraints as target.
2. Graph patterns describe fragments of directed, typed graphs which represent a coherent unit of the model (e.g., a service with a specification). “Atomic units” of transformations will be encoded in such patterns.
3. Graph transformation provides a high-level rule and pattern-based manipulation language to implement basic mappings between graphs. See e.g. [18] for a detailed definition of the semantics of graph transformations.
4. Complex transformations can be assembled using Abstract State Machine [12] rules defined on graph patterns and transformation rules (e.g. “Create a port in a WSDL document for all ports of a service”).

6.3 Derived Deployment Descriptor for the eUniversity Case Study

Fig. 16 shows an extract of the deployment descriptor («*services.xml*») file of the *ApplicationCreation* service derived by our model transformation. This configuration file describes the security and reliable messaging characteristics of the

provided service. The generation of the actual XML document is based upon the source model of Fig. 7 and the transformation rules of Fig. 15.

This configuration file is an extract of a WS-Policy-compliant descriptor which can be parsed by any Web service stack implementation which adheres to WS-Policy, WS-Security and WS-ReliableMessaging standards (although a reference is included to Apache's Sandesha reliable messaging platform, no semantic restrictions apply to the policy). Parameters in the configuration file are related to one Web service port. Boolean attributes which are marked in the model as *true* will be mapped to policy elements while concrete values (such as retransmission interval) will be filled with the specified value, respectively. Some technical details have been suppressed (such as schema URI). (*ExactlyOne* here refers to the policy semantics and not the messaging mode.) Note that this target language is extensible (e.g. logging can added easily) due to the nature of WS-Policy.

Note that according to the implementation of WS-Security standard (Rampart module), if a service is available via a secure connection, it cannot be accessed in plain text mode, moreover, the security settings (e.g. authentication token) of a port must be fixed. This implies that for clients with different security requirements, the service should be available at different URIs. Currently, the transformation creates a separate URI for every client-server (participant) pair with non-functional specifications.

```
<?xml version='1.0'?>
<service name="ApplicationValidationService">
  <operations>
  </operations>
   <wsp:Policy wsu:Id="ApplicationValidationServiceSecurityPolicy"
         xmlns:wsu="http://docs.oasis-open.org/wss/
           2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
         xmlns:wsp="http://schemas.xmlsoap.org/ws/
           2004/09/policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:Authentication
            xmlns:sp="http://schemas.xmlsoap.org/ws/
              2005/07/securitypolicy">
          <wsp:Policy>
            <wsp:authToken>
              <wsp:Policy>
                <sp:Username/>
              </wsp:Policy>
            </wsp:authToken>
          </wsp:Policy>
        </sp:Authentication>
        <sp:Encryption ...
          <wsp:Policy>
            <wsp:encryptBody/>
            <wsp:encryptAlgorithm>
              <wsp:Policy>
                <sp:Default/>
              </wsp:Policy>
            </wsp:encryptAlgorithm>
          </wsp:Policy>
        </sp:Encryption>
        <sp:DigitalSignature ...
          <wsp:Policy>
            <wsp:signBody/>
            <wsp:signAlgorithm>
              <wsp:Policy>
                <sp:Default/>
              </wsp:Policy>
            </wsp:signAlgorithm>
          </wsp:Policy>
        </sp:DigitalSignature>
        <sp:Timestamp ..
          <wsp:Policy>
            <wsp:useTimestamp/>
          </wsp:Policy>
        </sp:Timestamp>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  <wsp:Policy wsu:Id="ApplicationValidationServiceRMPolicy"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401
    -wss-wssecurity-utility-1.0.xsd"
    xmlns:wsrm="http://ws.apache.org/sandesha2/policy">
     <wsp:ExactlyOne>
       <wsp:All>
         <wsrm:filterDuplicates>true</wsrm:filterDuplicates>
         <wsrm:needsAck>true</wsrm:needsAck>
         <wsrm:maxNumberOfRetrans>3</wsrm:maxNumberOfRetrans>
         <wsrm:retransInterval>10000</wsrm:retransInterval>
         <wsrm:timeout>60</wsrm:timeout>
       </wsp:All>
     </wsp:ExactlyOne>
  </wsp:Policy>
</service>
```

Fig. 16 Fragments of services.xml of ApplicationCreation service

7 Related Work

The development of service-oriented systems has recently gained a lot of attention, and several approaches for modeling, generating and analyzing these software systems have been published or announced. However, most of these approaches focus mainly on functional requirements of SOS while non-functional aspects are neglected. We present the related work grouping them into the topics of modeling, performance analysis and deployment techniques.

7.1 UML Modeling Approaches

Several other attempts exist to define UML extensions for service-oriented systems and some approaches also are used for the automated transformation from UML to BPEL. Most of them, however, do not cover all three aspects types of model elements for structural, behavioral and non-functional aspects of SOAs. For example the UML 2.0 profile for software services [26] provides an extension for the specification of services addressing only structural aspects. Similarly, the current version of the UML profile and metamodel for services (soaML) [40] supports the structural concepts of service components, service specifications, service interfaces and contracts for services. soaML is the result of the standardization efforts started by the OMG in 2006. The UML extension for service-oriented architectures described by Baresi et al. [10] focuses mainly on modeling SOAs by refining business-oriented architectures. The refinement is based on conceptual models of the platforms involved as architectural styles, formalized by formal graph trans-

formation systems. The extension is also limited to stereotypes for the structural specification of services.

Other modeling approaches require very detailed UML diagrams from designers trying to force service-oriented languages (like BPEL) on top of UML in order to facilitate automated transformation from UML to BPEL. For example, the work of Skogan et al. [24] has a similar focus to our approach, i.e. a model-driven approach for services based on UML models. However, the approach lacks an appropriate UML profile preventing building models at a high level of abstraction; thus producing overloaded diagrams. Another example is the very detailed UML profile [6] that introduces stereotypes for almost all BPEL 1.0 activities - even for those already supported in plain UML, which makes the diagrams drawn with this profile hard to read. Some other extensions do not cover vital parts of service orchestrations such as compensation handling, e.g. the UML profile described in [32]. In a recently published article, Ermagan and Krüger [19] extend the UML2 with components for modeling services. Collaboration and interaction diagrams are used for modeling the behavior of such components. Neither compensation nor exception handling is explicitly treated in this approach.

Approaches addressing modeling of non-functional properties of services are quite rare. Examples are the OMG MARTE profile [39], and the extension proposed by Wada et al. [47], but conversely to the profile we presented in this work, none of them provides a "per contract" approach. Conversely to these approaches, UML4SOA(-NFP) focuses on the improvement of the expressive power of UML by defining a small set of stereotypes for structural and behavioral as-

pects of SOAs, focusing on service-oriented features as orchestrations and the non-functional aspects of service-oriented systems as shown in this article. For a more thorough discussion of UML4SOA, see [20].

7.2 Methods for Analyzing Non-Functional Properties

Performance evaluation of software models has gained increased attention over the last decade (see [9] for a review of this field). Given the centrality of the UML, many approaches have dealt with the extraction of performance models from activity diagrams [31, 14], sequence diagrams [11] and state machine diagrams [35]. The use of an intermediate meta-model to facilitate these translations has been proposed in [49, 41], in which concrete application to layered queueing networks and stochastic Petri nets have been given. A work closely related to ours is [16], in which the performance prediction of service compositions is carried out on BPEL models. A BPEL workflow is expressed as a single annotated activity diagram, which is translated into a layered queueing network for the analysis. The semantics of the translation and the profiles used for the performance annotations are very similar, however our work extends the scope of applicability of performance prediction to a more general scenario in which interdependency between orchestrations is taken into account.

Alternatively to formal analysis models, the dependability and robustness of services can be also investigated by using fault injection techniques as discussed in [30]. The authors of [29] use monitoring and testing techniques to evaluate the dependability of web services by using statistical real-time data. [5] aims to de-

velop a dependable web services framework, which relies on extended proxies. However, this needs a modification at the client side in order to handle exceptions and find new service instances. Moreover, the reconfiguration of client side proxies uses non-standard WSDL extensions while we concentrated on standards-compliant solutions.

7.3 Deployment Mechanisms for Non-Functional Properties

A framework for automated WSDL generation from UML models is described in [44], using the UML extensions of MIDAS [13]. In [23], web service descriptions are mapped to UML models, and (after using visual modeling techniques) a composite service can be created for which the descriptor is automatically generated. However, none of these works considers non-functional properties of web services.

Non-functional aspects of e-business applications are discussed among others in [8], having some description of deployment optimization for J2EE applications, but without discussing details of model-based deployment. Integration of non-functional aspects in the development by model transformations is also investigated in [15,42] and [27], focusing on parts of the engineering process, although using different underlying transformation techniques for model analysis and deployment. An early version of the deployment transformation suite was presented in [21].

8 Conclusions and Future Work

Despite the advantage of coherent, separable components with well-defined interfaces, service-oriented systems can become as complex as any other. For this reason, model-driven development is invaluable in the creation and maintenance of service-oriented systems. High-level models allow us to retain intellectual control of complex systems which would otherwise defeat our attempts to understand them in either static or dynamic terms. Making these models an integral part of the development process means that they grow and change as the system grows and changes and they are available to support the extension and adaptation of the system in response to perceived need or demand.

While model-driven development has gained great acceptance in documenting the static structure of systems in terms of components, packages, classes and interfaces, modeling of functional properties has received less attention and modeling of non-functional properties has received much too little. Non-functional properties such as responsiveness, availability, scalability and security have a direct impact on whether the system is accepted and valued by end users. In contrast, the internal organization of the codebase into packages and classes is entirely invisible and irrelevant to end users. From this perspective the current emphasis on modeling of static software structure seems misplaced, to say the least.

In this paper we have presented a model-driven approach for the development of service-oriented systems with explicit support for the specification of non-functional properties. Our main contributions are

- the ability to specify non-functional properties right within the model of the SOA system, enabling modeling of performance and security,
- model-based support for performance analysis, in particular performance estimates and reliability analysis, based on the timed process algebra PEPA,
- and the introduction of deployment mechanisms that comprise model-to-model and model-to-text transformations.

We also created a method for analyzing the performability-reliability vs. performance of services with non-functional parameters as described in [22], but which is not included in this work.

We plan to extend the current UML4SOA-NFP approach to cover system's reliability at architecture level. Future work will necessarily encompass further validation of the approach presented against larger projects. We plan to apply it to more complex case studies in collaboration with industry.

References

1. Software Engineering for Service-Oriented Overlay Computers. http://www.sensoria-ist.eu.
2. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). http://docs.oasis-open.org/wss/v1.1/.
3. WS-Reliability 1.1 specification. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf.
4. WS-ReliableMessaging 1.1 specification. http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf.

5. E. Alwagait and S. Ghandeharizadeh. DeW: A Dependable Web Services Framework. *RIDE*, 01:111–118, 2004.
6. J. Amsden, T. Gardner, C. Griffin, and S. Iyengar. Draft UML 1.4 Profile for Automated Business Processes with a Mapping to BPEL 1.0. Specification, IBM, 2003. `http://www.ibm.com/developerworks/rational/library/content/04April/3103/3103_UMLProfileForBusinessProcesses1.1.pdf`, Last visited: 10.12.2008.
7. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004.
8. A. Balogh, D. Varró, and A. Pataricza. Model-Based Optimization of Enterprise Application and Service Deployment. In *ISAS*, pages 84–98, 2005.
9. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.
10. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-Based Modeling and Refinement of Service-Oriented Architectures. *Journal of Software and Systems Modeling (SOSYM)*, 5(2):187–200, 2005.
11. S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In P. Inverardi, S. Balsamo, and B. Selic, editors, *Proceedings of the Third International Workshop on Software and Performance*, pages 35–45, Rome, Italy, July 2002. ACM.
12. E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
13. P. Caceres, E. Marcos, and B. Vera. A MDA-Based Approach for Web Information System Development. In *Workshop in Software Model Engineering (WiSME@UML2003)*,

2003.

14. C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens. Analysing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. In Dujmovic et al. [17], pages 74–78.
15. V. Cortellessa, A. D. Marco, and P. Inverardi. Software performance model-driven architecture. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1218–1223, New York, NY, USA, 2006. ACM Press.
16. A. D'Ambrogio and P. Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 78–89. ACM, 2007.
17. J. J. Dujmovic, V. A. F. Almeida, and D. Lea, editors. *Proceedings of the Fourth International Workshop on Software and Performance, WOSP 2004, Redwood Shores, California, USA, January 14-16, 2004*. ACM, 2004.
18. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
19. V. Ermagan and I. Krüger. A UML2 Profile for Service Modeling. In *International Conference on Model Driven Engineering Languages and Systems*, volume LNCS 4735 of *IEEE*, pages 360–374. Springer-Verlag, 2007.
20. H. Foster, L. Göczy, N. Koch, P. Mayer, C. Montangero, and D. Varró. D1.4b: UML for Service-Oriented Systems. Specification, SENSORIA Project 016004, 2010.
21. L. Gönczy, J. Ávéd, and D. Varró. Model-based deployment of web services to standards-compliant middleware. In P. Isaias, M. B. Nunes, and I. Martinez, editors, *Proc. of WWW/Internet 2006(ICWI2006)*. Iadis Press, 2006.
22. L. Gönczy, Z. Déri, and D. Varró. Model-Based Performability Analysis of Service Configurations with Reliable Messaging. In N. Koch, A. Vallecillo, and G.-J. Houben,

editors, *Proc. Model Driven Web Engineering (MDWE)*, CEUR Vol-389, 2008.

23. R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-Driven Web Services Development. In *Proc. of the IEEE Int. Conf. on e-Technology, e-Commerce and e-Servie (EEE'04)*, pages 42–45, Los Alamitos, CA, USA, 2004. IEEE.
24. R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Style-Based Modeling and Refinement of Service-Oriented Architectures. In *Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04)*, IEEE, pages 47–57. IEEE, 2004.
25. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
26. S. Johnson. UML 2.0 Profile for Software Services. Specification, IBM, 2005. `http://www.ibm.com/developerworks/rational/library/05/419_soa`, Last visited: 10.12.2008.
27. H. Jonkers, M.-E. Iacob, M. M. Lankhorst, and P. Strating. Integration and Analysis of Functional and Non-Functional Aspects in Model-Driven E-Service Development. In *EDOC*, pages 229–238, 2005.
28. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling*. Wiley-Interscience, IEEE Computer Society, 2008.
29. P. Li, Y. Chen, and A. Romanovsky. Measuring the Dependability of Web Services for Use in e-Science Experiments. In D. Penkler, M. Reitenspieß, and F. Tam, editors, *Service Availability, Third International Service Availability Symposium, ISAS 2006, Helsinki, Finland, May 15-16, 2006, Revised Selected Papers*, volume 4328 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2006.
30. N. Looker and J. Xu. Dependability Assessment of Grid Middleware. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 125–130. IEEE Computer Society, 2007.

31. J. P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. In Dujmovic et al. [17], pages 25–36.

32. K. Mantell. From UML to BPEL. Specification, IBM, 2005. http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel/, Last visited: 10.12.2008.

33. P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *SCC'08*, IEEE, pages 1–6. IEEE, 2008.

34. P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *The 12th IEEE International EDOC Conference (EDOC 2008)*, pages 203–212, Munich, Germany, 2008. IEEE Computer Society.

35. J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In M. Silva, A. Giua, and J. Colom, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

36. Object Management Group. *UML Profile for Schedulability, Performance and Time Specification*, 2005. http://www.omg.org/technology/documents/formal/schedulability.htm.

37. Object Management Group. *UML for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, v1.1* , 2008. http://www.omg.org/spec/QFTP/1.1/.

38. Object Management Group (OMG). Unified Modeling Language: Superstructure, version 2.1.2. Specification, OMG, 2007. http://www.omg.org/docs/formal/07-11-02.pdf.

39. Object Management Group (OMG). UML Profile for MARTE, Beta 2. Specification, OMG, 2008. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf.

40. Object Management Group (OMG). Service Oriented Architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS), revised submission. Specification, OMG, 2009. http://www.omg.org/cgi-bin/doc?ptc/09-04-01, Last visited: 30.08.2009.

41. D. Petriu and C. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Softw. Syst. Model.*, 6:163–184, 2007.

42. S. Röttger and S. Zschaler. Model-Driven Development for Non-functional Properties: Refinement through Model Transformation. In *Proc. The Unified Modeling Language (UML 2004)*, volume 3273 of *LNCS*, pages 275–289. Springer, 2004.

43. M. Tribastone and S. Gilmore. Automatic Extraction of PEPA Performance Models from UML Activity Diagrams Annotated with the MARTE Profile. In *Proceedings of the Seventh International Workshop on Software and Performance (WOSP)*, Princeton, New Jersey, USA, June 2008. ACM.

44. J. M. Vara, V. de Castro, and E. Marcos. WSDL Automatic Generation from UML Models in a MDA Framework. In *NWESP 2005*, page 319. IEEE, 2005.

45. D. Varró and A. Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, October 2007.

46. D. Varró and A. Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 290–304, Lisbon, Portugal, October 10–15 2004. Springer.

47. H. Wada, J. Suzuki, and K. Oba. Modeling Non-Functional Aspects in Service Oriented Architecture. In *IEEE International Conference on Service Computing, Chicago IL*, pages 222–229. IEEE, 2006.

48. M. Wirsing, M. Hölzl, L. Acciai, A. Clark, F. Banti, A. Fantechi, S. Gilmore, S. Gnesi, L. Gönczy, N. Koch, A. Lapadula, P. Mayer, F. Mazzanti, R. Pugliese, A. Schroeder, F. Tiezzi, M. Tribastone, and D. Varró. A Pattern-Based Approach to Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In *Proc. of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2008), Porto Sani, Greece*, LNCS. Springer-Verlag, 2008.

49. C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Meseguer. Performance by unified model analysis (PUMA). In *WOSP*, pages 1–12. ACM, 2005.